# Programming Aided by Machine Learning

Ollie Willette
olipatw@gmail.com
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA

## Abstract

The task of writing correct code efficiently is a large problem in computer science, and is interesting from theoretical and practical perspectives. We look at how recent advancements in machine learning are being used to aid in code writing, and discuss possible future places to research. We also explore how the idea of using machine learning to aid in code writing is related to the task of program synthesis, but in contrast to program synthesis, we expect that there needs to be human intervention at multiple levels of program development.

***Keywords:*** codex, copilot, alphacode, machine learning

## 1 Introduction

The task of writing correct programs efficiently is interesting, especially to people who spend much of their time writing programs. By correct we simply mean programs that do what we want without crashing. From a theoretical perspective, this task is complex, as which parts of programming can and cannot be automated is unclear, and what automating different parts of programming would look like is worth discussing. From a practical perspective, this is worth considering because programmers spend lots of time programming. Therefore it is important that they waste as little time as possible, and use all tools available.

One distinction that should be emphasised is where the problem of programming aid ends and program synthesis starts. There is a lot of overlap between the two fields; the two fields can work with one another when appropriate, but they are distinct. The goal of program synthesis is to develop machines that can automatically generate programs to solve a given problem, solely based on the initial problem description without requiring any additional human input. Historically program synthesis was based on a high-level formal specification of the problem. This formal specification was found to be as hard to produce as the program code itself [2], and the field of program synthesis moved to less strict specifications.

The goal of programming aids is finding the parts of programming that can be automated or made easier. Making programming easier is an important thing to remember in this paper; we don't expect machine learning to write entire programs for us.

Machine learning is when computer programs are able to learn from data, using algorithms and statistics to "analyze" it. This paper will be looking at how machine learning is being used to aid in programming. Ideally the programmer describes the solution to the problem in natural language, and has some program (e.g. a generative pretrained transformer) that handles the ambiguity of natural language to generate the best guess of what the programmer wants. The programmer then engages with the output to make a correct solution to that problem.

Certain difficulties of programming that can be mitigated will be looked at in Subsection 2.1. Recent advances in machine learning [3] have opened up new methods of investigation with regards to programming aids. We will go into the basic principles of machine learning in subsection 2.2. After that, in Section 3 we will look at how Kulal [4] tried to reduce the problem of programming to writing fine grain pseudocode. Then we will talk about how recently OpenAI [1] created Codex, based off of GPT-3, to take a high-level description of the solution to a problem and try to generate the correct code. From there we branch into Google's AlphaCode [5] where Google created a machine quite similar to Codex, but with the exclusive task of solving competitive programming problems. The other branch will have us looking at an evaluation of Copilot [9], which is GitHub's programming tool implemented with Codex. Then we go to our conclusion in Section 4, discussing possible future directions.

## 2 Background Information

### 2.1 Programming and Syntax

Many difficulties arise when programming, and it is conducive to the practice of computer science to lessen these difficulties. One difficulty in programming worth highlighting for this paper is syntax. Sometimes a programmer knows what they want a program to do, but are not certain about the syntax to do it. For example, if a programmer has an array, and they want to know how long the array is, the programmer just has to memorize a table similar to 1, but for all languages and units the programmer uses.

Let us consider a slightly larger task, of reading from a file, capitalizing the alphabetic letters, and printing the resulting text. Although this task may not be considered complex, the syntax required to accomplish it may differ significantly between programming languages. Additionally, the task description is sufficiently clear to be understood by a computer [12]. Despite this, a programmer may encounter

**Table 1.** Retrieving Size of a Collection in Various Languages

| Language | Unit | Method |
|---|---|---|
| Java | Standard array | .length |
| Java | String | .length() |
| Java | ArrayList | .size() |
| Clojure | Any collection | (count ) |
| Python | Any collection | len() |
| Lua | Array | getn() |
| PHP | Array | sizeof()* |
| C++ | Vector | .size() |
| C | Array | prayer |

* sizeof() is also a function in C that does something different from what it does in PHP

difficulty in translating this description into code, since no programming language would accept the description as is. It is reasonable to question why a programmer should have to spend time deciphering the necessary code when an explicit description of the desired behavior has already been provided. Ideally the written-out solution to a programming problem should be expressed as simply as possible. The programming language Python is a good example of one way to make programming simpler, and Python does a good enough job, removing a lot of bloat that is in a lot of other programming language's syntax. Consider the example of opening a file and printing all of it capitalized to stdout in C++, slightly modified from [8]

```
std::ifstream myfile("demofile.txt");
while (myfile) {
    std::cout << std::toupper(myfile.get());
}
```

versus in Python [11]

```
f = open("demofile.txt", "r")
print(f.read().upper())
```

In Python, there is still room for error in knowing that the *open*, *read*, and *print* functions exist, how they work, and how to use the functions, but it is more concise than C++.

Issues like knowing the names to functions and when to use them are limitations to all programming languages, as programming languages must have specific, non-ambiguous syntax. This is because programming languages must describe the exact instructions for the compiler/interpreter, as any deviation would be harmful to the process of programming. It is inefficient for programmers to be looking at compiled code to make sure the compiled code is actually correct, so programming languages can have no ambiguity in what a statement accomplishes functionally. Natural language, though, has lots of ambiguity, and much more complicated context, but every person is familiar with a natural language, and can use natural language much more efficiently. The issue was that computers were not capable of processing natural language effectively, but this has changed with recent developments in the field of machine learning [10].

## 2.2 Machine Learning

The recent advances with regards to programming we will be looking at all have to do with machine learning, and the newest use large language models (LLM). LLMs are language models built on neural networks with billions of parameters, and trained on huge amounts of data. A language model is a program that gives the probability that a given sequence of words appears in the language. The challenge for language models lies in the infinite amount of possible sequences in a language. The two LLMs of note in this paper are GPT-3 [1] and AlphaCode [5] which are both transformers that get pretrained on unlabeled data. Data, and unlabeled data, will be talked about later in the section.

In their research of mapping pseudocode to compilable code Kulal and Pasupat, unlike Codex and AlphaCode, did not use a transformer, but a long short-term memory [4]. LSTMs were state of the art with regards to natural language processing until transformers were introduced [10].

Transformers and LSTMs are both based on the idea of a neural network, which is a type of machine learning model which consists of layers of interconnected nodes, or "neurons," that process information and make predictions based on input data. There are also values between the layers called weights, that affect how the data moves from one layer to the next. Weights are then changed during the training process.

A fundamental part of using neural networks is training data. Training data for neural networks typically consists of input and output pairs. (e.g, for the problem of determining what animal is in a picture, an input/output pair could be a picture of a cat and "cat".) The input represents the data that is fed into the neural network, while the output represents the expected result of the network's computation given that input. Learning is done by running a large amount of training data though the model, and taking the generated output. Then the generated output is compared against the correct answer, and taking the difference, which is called the error. The error gets propagated backwards through the layers, affecting the weights (values between layers) in order to reduce the future error. The methods of changing the weights is all based on ideas from statistics such as regression analysis.

Recurrent neural networks (RNNs) are a type of neural network that can read and write sequential data by using loops in the network, which allows information to persist across time steps. This enables recurrent neural networks to capture dependencies and patterns in sequences, such as in natural language processing and speech recognition. But RNNs have an issue called the vanishing gradient problem. The vanishing gradient problem is where as input is fed through the model, older input becomes less and less relevant, quickly "disappearing," which can negatively affect the model's ability to succeed.

Long short-term memory (LSTM) is a subtype of recurrent neural network, designed to address the vanishing gradient problem. LSTMs try to solve this by using gated memory units that can selectively remember or forget information over time, allowing them to maintain long-term dependencies and make accurate predictions on longer sequences.

For a clear and full explanation of transformers see [10], but for this paper, understand that a transformer is a machine learning model, using neural networks, that can handle sequences of data like RNNs, but without using loops inside of the network. Furthermore, transformers have attention mechanisms, that are nodes dedicated to looking at the previous data in a sequence and determining which parts of the sequence are important. Transformers are used to build Codex, GPT-3, and AlphaCode.

Codex, GPT-3, and AlphaCode all get "pretrained." Pretraining is the idea of giving training data which is not necessarily representative of what you want the neural network to do, but similar enough to be valuable. For example, GPT-3 was pretrained on data from all over the internet, with the goal of just predicting the next word to appear. Because the data was pulled from all around the internet, there was little known about the data, making it "unlabeled" data, as apposed to labeled data, where the correct output and more is known. This training, on unlabeled data, made GPT-3 very effective at the task of natural language processing and generation. This is where the name GPT comes from; Generative, Pretrained, Transformer.

## 3 Research

### 3.1 Pseudocode to compilable code

Kulal and Pasupat [4] in 2019 did research on the topic of mapping pseudocode to compilable code. This is in the gray area between program synthesis and programming aid, and can very well be considered both. The work of Kulal and Pasupat involved programs of about 15 lines of code in C++; they then wrote pseudocode to describe each line of code with high specificity. For example the pseudocode

read n values into array a and array b

would be used for the line of code

```
for(int i = 0; i < n; i++) cin >> a[i] >> b[i];
```

The corpus of pseudocode was written by over 50 different people, as Kulal and Pasupat made use of Amazon's "Mechanical Turk" to have help writing the pseudocode for a line of code. The workers had no formal qualifications, but were tested to provide acceptable pseudocode for some code. Those that passed the test then were allowed to work on the project. Using the pseudocode and its associated code as the training dataset, Kulal and Pasupat trained a LSTM network to write a new program given a new pseudocode description.

Checking if code is correct is a difficult task, as surface level metrics like exact sequence matching can fail for functionally equivalent lines of code [1, 4, 5]. For example the lines

```
if(a){run();}
if(a==true){run();}
if(!a){}else{run();}
```

all have the same functionality despite looking different. Kulal and Pasupat looked to the program synthesis community, where the notion of functional correctness is the gold standard. Checking functional correctness means that the program generates the correct output for test input. The tests were written by Kulal and Pasupat. One downside to testing functional correctness is that checking code requires running the generated code, whereas sequence matching can be done without running code. Kulal and Pasupat further implemented pass@k, meaning the framework generates k samples and considers the problem solved if any of the k samples successfully pass all tests [4]. This was new to the research community, and has been continued in use by many researchers, including OpenAI and DeepMind. Kulal and Pasupat used 100 samples to test if a problem was solved, eventually using compiler errors of early samples to automatically improve future samples for a given problem. They did this by using a second neural network to predict the offending line, and replace it with a new line of code. This is analogous to what a human programmer does, since human-written code rarely compiles flawlessly the first time.

Kulal and Pasupat eventually got to a model that passed 44.7% [4] of the test cases, which is impressive, but not paradigm shifting.

### 3.2 Codex and OpenAI

OpenAI is the research laboratory responsible for creating the GPT family. They recently had a team of researchers develop a machine learning model based on GPT-3, called Codex. Codex was trained on 159 GB of Python code from GitHub and was developed to take a function declaration and docstring (a comment at the beginning of a function describing what the function does) and produce a working function in Python. Codex was developed in the pursuit of program synthesis, but is now the core of GitHub Copilot, a programming aid. Describing each function is a much higher level of granularity compared to the work of Kulal and Pasupat, which described every line of code. The docstrings were about 1-4 sentences long. e.g. [1]

```
def vowels_count(s):
"""Write a function vowels_count which takes a
string representing
a word as input and returns the number of vowels in
the string.
Vowels in this case are 'a', 'e', 'i', 'o', 'u'.
Here, 'y' is also a
```

```
vowel, but only when it is at the end of the given
word.
Example:
>>> vowels_count("abcde")
2
>>> vowels_count("ACEDY")
3"""
```

For which Codex might generate the (incorrect) solution:

```
vowels = "aeiou"
v = 0
for i in s.lower():
    if i in vowels:
        v += 1
return v
```

This example fails, as it will not count ending ys as vowels. Note the docstring gives two examples, but Codex does not have any mechanism for separating out and specifically using that information.

The final model of Codex has 12 billion neural network parameters (basically the weights between the layers). Codex achieved successful completion of 70.2% of the problems it was tested on, which is a large improvement over the work of Kulal and Pasupat. The test set was called HumanEval [1] which was created by OpenAI because any pre-existing programming problem set (i.e, Codeforces) would have problem solutions already existing on GitHub. Problems that have already been solved on GitHub shouldn't be tested on, as that would just allow Codex to copy and paste solutions it had seen during training. Furthermore, GitHub is known to have malicious programs (i.e. programs that damage the computer when run) [1], so the OpenAI team ran the programs in an isolated and safe environment. Codex was tested further on competitive programming problems pulled from around the internet, where it achieved a solution for 3.08% of the 10,000 problems. These low numbers are despite the fact that Codex might have been trained on the solutions to these competitive programming problems. Codex is far from replacing programmers, and the biggest issue was succinctly stated by one of the researchers at OpenAI

> We find that as the number of chained building blocks [e.g. instructions] in the docstring increases, model performance decreases exponentially. This behavior is uncharacteristic of a human programmer, who should be able to correctly implement a program for a chain of arbitrary length if they can do so for a chain of length two.[1] [1]

[1]This may be solved in Auto-GPT [7], which is able to make prompt calls to itself, and for complex problems will now often prompt itself to break the complex problem into smaller problems, and then prompt itself on the smaller problems, and then prompt itself to join them together. More research is needed on this.

There are more issues with Codex, such as misalignment. Misalignment refers to the phenomenon where a "request" (e.g. the description of a function) is made to the LLM to perform a certain task (write the function), but it attempts to perform a different one. Misalignment is possible because language models like Codex are based on statistical patterns learned from large data sets of text. While they can generate impressive outputs, they do not truly "understand" language in the way that humans do. They are simply applying statistical patterns and making predictions based on the input they receive. This means that there is always a possibility for the model to misinterpret or misunderstand the input, leading to misaligned outputs.

Another issue with Codex, and all LLMs, is bias. This is because LLMs are trained on large data sets that have biases present in the data sets. These biases may be unintentionally introduced through the collection and curation of the data, or they may reflect societal biases that exist in the real world. When an LLM is trained on biased data, it can learn to perpetuate and amplify these biases in its outputs. For example when I gave Codex the code fragment

```
# sql insert for person name Tyrone Williams
INSERT INTO employeeList(name, startdate, salary,
hourlyRate, hoursPerWeek, position, manager)
```

it generated the completing code

```
VALUES ('Tyrone Williams', '2020-04-15',
'$50,000', '$20.00', '40', 'Accountant',
'Jane Doe');
```

this is exactly what a SQL programmer would expect to come next, but the format, data type, and value for each entry is random, as there is no correct answer. Then done again, using the same input with a change in name

```
# sql insert for person name Andrew Hill
INSERT INTO employeeList(name, startdate, salary,
hourlyRate, hoursPerWeek, position, manager)
```

creates the new code

```
VALUES ('Andrew Hill', '2019-01-01',
50000, 25, 40, 'Software Developer',
'John Smith');
```

The fictional character, Andrew Hill, has worked for the company longer and is reported to earn $5 more per hour than Tyrone Williams. This is an example of bias, which was mentioned in both [1, 5]. It is important to note that this information was derived from an initial test of OpenAI's Codex API [6] using default settings, which may be subject to random fluctuations and should be interpreted with caution.

Note that despite the same hours per week worked and different hourly pay, they receive the same annual salary which is logically inconsistent.

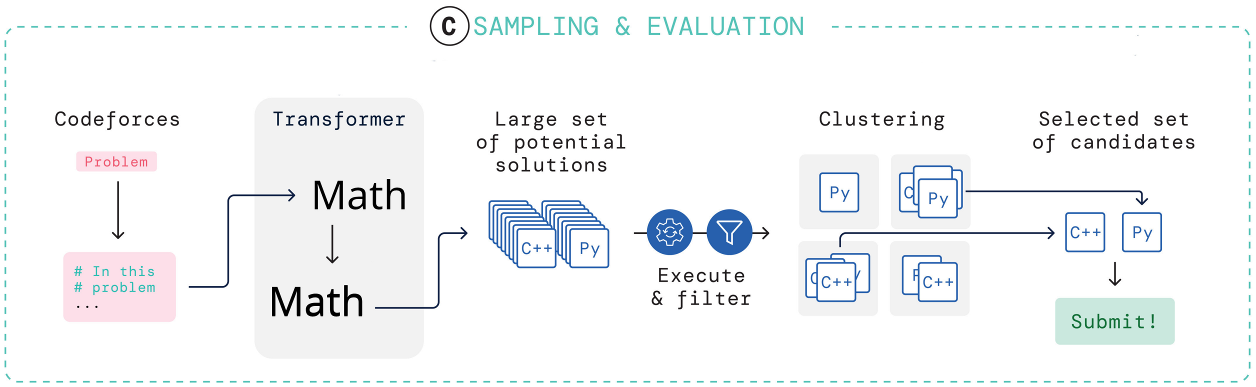

**Figure 1.** AlphaCode filtered from given tests, then executed the programs on the synthetic input and grouped programs based on output. [5]

## 3.3 AlphaCode

DeepMind, a research laboratory owned by Google, created AlphaCode to solve competitive programming problems. AlphaCode has 41 billion parameters, compared to Codex's 12 billion. AlphaCode was pretrained on 715 GB of GitHub code in nearly all programming languages. All of the data AlphaCode trained on was released before the creation of the problems it was tested on, making sure that it was not just copying previous answers to the problem. The capability of AlphaCode to access prior solutions to past problems is analogous to the ability of competitive programmers to review previous year's problems and solutions before participating in a competition.

AlphaCode was then fine-tuned and tested using 2.6 GB of code curated by DeepMind called CodeContests. Fine-tuning meant the data had a higher weight of importance. The fine-tuning data also had metadata that might be helpful, such as difficulty rating, or tags that indicate a path to the solution like "greedy" or "dynamic programming". This metadata is not available at contest time, so AlphaCode could not see the metadata from the problems it was tested on, just the problems it was fine tuned on. Being fine tuned on 13328 of the problems, and tested on the other 165 problems, AlphaCode was able to solve 29.6% of those 165 [5]. As shown in figure 1, we see a technique they used to improve AlphaCode's chances of success. This technique was to have AlphaCode search through the problem statement and look for test examples, e.g. in the section 3.2 we gave the example problem of vowels_count. Inside of the problem statement there were test cases given, which AlphaCode, unlike Codex, would then interpret into tests such as

```
assert(vowels_count("abcde") == 2)
assert(vowels_count("ACEDY") == 3)
```

Then, when solving that problem, AlphaCode would generate tens of thousands of sample solutions for just the one problem. One of thousands of problems generated might be exactly like the incorrect solution Codex generated in 3.2, and another generated might be the same but with

```
if s.lower()[-1] == 'y':
    count += 1
```

before the return statement. AlphaCode will then run all sample solutions against the tests generated from the given examples. This would mean that both hypothetical potential solutions would pass the first test, but only the second sample solution would pass the second test. (Because only the second solution would count the trailing y as a vowel.) So the first sample solution would be thrown away. This process of filtering filtered out 99% of sample solutions generated by AlphaCode.

The team at DeepMind then developed a second, specialized transformer model to generate synthetic test inputs for a given problem and its example inputs. For instance, in the problem of count_vowels, the synthetic test inputs may generate strings such as "abcda", "AB", or even something like "123", and "H as d fa". These synthetic inputs are then given to the remaining sample solutions, which have already passed the example tests. The programs that produce the same output for the synthetic inputs are grouped together. Then they evaluated the group with the most sample solutions as the best group. This is on the logic that there is one correct way to solve a problem, but many incorrect ways.

## 3.4 Copilot Evaluation

In [9] Vaithilingam et al. conducted a within-subjects user study with 24 participants to understand how programmers use Copilot, a programming aid based on Codex, a LLM made with the goal of program synthesis. Copilot is much like what was described in Section 1: a machine that helps a programmer by producing an attempted solution to a problem that the programmer then improves on to make a correct solution.

**Table 2.** Copilot vs Intellisense times

| | Task 1 - Easy | | Task 2 - Medium | | Task 3 - Hard | |
|---|---|---|---|---|---|---|
| | Intellisense | Copilot | Intellisense | Copilot | Intellisense | Copilot |
| | 9 : 35 | 1 : 46 | 7 : 48 | 12 : 53 | 13 : 41 | 11 : 08 |
| | 3 : 50 | 3 : 57 | 15 : 52 | 16 : 45 | 13 : 43 | 11 : 05 |
| | 4 : 49 | 4 : 55 | 16 : 28 | 7 : 26 | 22 : 42 | 4 : 04 |
| | 9 : 04 | 6 : 18 | 14 : 16 | 15 : 05 | 13 : 06 | DNF |
| | 5 : 18 | 1 : 18 | 7 : 35 | 13 : 24 | 23 : 13 | 19 : 54 |
| | 15 : 54 | 7 : 52 | 12 : 39 | DNF | 4 : 48 | DNF |
| | 5 : 27 | 3 : 12 | 10 : 47 | 6 : 02 | DNF | DNF |
| | 2 : 09 | 20 : 12 | 8 : 30 | DNF | DNF | 9 : 19 |
| Average Time | 7 : 01 | 6 : 11 | 11 : 44 | 11 : 56 | 13 : 36 | 11 : 06 |
| Overall average time for all tasks combined | | | | | 10 : 23 | 9 : 18 |

Table 1: Individual and average task completion times. DNF implies the participant did not finish in 25 minutes. [9]

The study had users using either Copilot or Intellisense, the standard auto-complete that most programmers are familiar with that is by default pre-installed and enabled for most IDEs. This decision creates two potential issues. First, programmers are much more familiar with Intellisense than Copilot, as Intellisense has been available and used for over 20 years, and Copilot is less than 2 years old, and very rarely used, at the time of research. Furthermore, Intellisense and Copilot are by no means mutually exclusive, and do different things. So while users might be familiar and reliant on Intellisense, they are unfamiliar and confused by Copilot.

The users within the study had a range of experience with programming, from 2 to 5+ years of experience. One of the users was a software developer, the rest were students from undergraduate to Ph.D. level. The study suggests Copilot does not improve task completion time or success rate on average. Users generally reported liking Copilot despite this, saying it lead to less time searching online [9].

The exact times for the participants can be seen in Table 2 for exact task completion times. Note that of the 24 users, 8 were selected to do each task, which they would then do using one of the tools (either Copilot or Intellisense, randomly chosen) then the same user would complete the same task with the other tool. The easy task was read a csv file, and remove the first and last elements. The medium task was scrape a webpage, and extract the hyperlinks, writing them to a file. The hard task was read a csv file, and convert it to a graph representing the data [9]. The average time using Copilot did not improve over using Intellisense. Furthermore, the completion rate decreased. The researchers commented

> participants encountered difficulties in understanding, editing, and debugging the code snippets generated by Copilot, which significantly hindered their task-solving effectiveness. [9]

One thing to note from Tabel 2 is that the fastest completion time for each task was always with Copilot. Furthermore, users reported preferring Copilot over Intellisense, although they trusted the code of Intellisense more than the code generated by Copilot.

## 4 Conclusion

In conclusion, recent advancements in machine learning may have an affect on the field of programming, and researchers are exploring the use of deep learning models to generate code. Kulal and Pasupat researched mapping pseudocode to compilable code using an LSTM, and their findings have contributed to the development of more sophisticated models like Codex. However, the use of these models is not without its challenges, like issues with not working code, bias, and misalignment, as the team at OpenAI found. Additionally, DeepMind's AlphaCode, improved in solving competitive programming problems through testing. Vaithilingam et al.'s study on Copilot, a programming aid based on Codex, suggests that programmers don't benefit greatly from the use of such tools, and those with less experience may struggle to make effective use of the tools.

It seems that more issues arise when people who are less familiar with coding rely on Copilot, but that gains might be had for people with a solid foundation of knowledge with regards to the programming task. Perhaps these issues were exacerbated by the newness of their experience with Copilot, and that with more familiarity with the tool more gains will be seen. More issues arise when the code generated by Copilot is incorrect. We saw that AlphaCode had big gains in code quality over OpenAI's Codex when they introduced a test to filter out incorrect code. So perhaps tools like Copilot could look into adding the ability for the user to add a test that the generated code should pass. A feature like this may be useful, but the implementation would need to be wary about potentially malicious code that needs to be sandboxed before run. This is an area to be researched further.

## Acknowledgments

Many thanks to Elena Machkasova, my advisor who suffered through all my faults valiantly. Also thank you to my friends who gave helpful feedback and moral support.

It should be known that this essay was written with the help of ChatGPT. For a full list of prompts/responses relating to this essay, see Programming Aided by Machine Learning ChatGPT prompt/response list

## References

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[2] Sumit Gulwani, Alex Polozov, and Rishabh Singh. 2017. *Program Synthesis*. Vol. 4. NOW. 1–119 pages. https://www.microsoft.com/en-us/research/publication/program-synthesis/

[3] M. I. Jordan and T. M. Mitchell. 2015. Machine learning: Trends, perspectives, and prospects. *Science* 349, 6245 (2015), 255–260. https://doi.org/10.1126/science.aaa8415 arXiv:https://www.science.org/doi/pdf/10.1126/science.aaa8415

[4] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. SPoC: Search-based Pseudocode to Code. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/7298332f04ac004a0ca44cc69ecf6f6b-Paper.pdf

[5] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158 arXiv:https://www.science.org/doi/pdf/10.1126/science.abq1158

[6] API OpenAI. [n. d.]. OpenAI API. https://platform.openai.com Accessed on April 10th, 2023.

[7] Significant-Gravitas. [n. d.]. Significant-gravitas/auto-GPT: An experimental open-source attempt to make GPT-4 fully autonomous. https://github.com/Significant-Gravitas/Auto-GPT Accessed on April 15th, 2023.

[8] Udacity. 2021. How to read from a file in C++. https://www.udacity.com/blog/2021/05/how-to-read-from-a-file-in-cpp.html Accessed on April 10, 2023.

[9] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the usability of code generation tools powered by large language models. *CHI Conference on Human Factors in Computing Systems Extended Abstracts* (2022). https://doi.org/10.1145/3491101.3519665

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

[11] w3schools. n.d.. Python file open. https://www.w3schools.com/python/python_file_open.asp Accessed on April 10, 2023.

[12] Ollie Willette. [n. d.]. Codex ReadCapitializePrint. https://platform.openai.com/playground/p/8Tt93Zb2q6dfw5Xtay9CAswM?model=text-davinci-003 Accessed on April 26th, 2023.